The 2010 IEEE/RSJ International Conference on
Intelligent Robots and Systems
October 18-22, 2010, Taipei, Taiwan

# Component-based Refactoring of Motion Planning Libraries

Davide Brugali, Walter Nowak, Luca Gherardi, Alexey Zakharov, Erwin Prassler

*Abstract*— **Most of the current state of the art motion planning software libraries are not easily interchangeable, because core concepts are represented with different data structures, application programming interfaces (API) are not compatible, or algorithms are encapsulated in modules organized in mutually exclusive abstraction hierarchies. These problems limit the possibility to reuse different libraries in the same application interchangeably and to compare their quality attributes (performance, completeness, etc.). An approach to overcome these shortcomings is refactoring, a technique that aims to restructure a set of existing software libraries without affecting their external behavior in order to harmonize their architecture, data structures, and APIs.**

**This paper presents a component-based refactoring approach that has allowed the transition from motion planning libraries, taking the object-oriented framework CoPP as basis, to a component-based system. In particular we describe a four-step application of well-known architecture refactoring patterns that redistributes the responsibilities among the classes, harmonizes the common data structures and reduces the coupling degree. The obtained system represents a composition of reusable components that are easy to customize and offer different algorithms to resolve the same problem. In this way the user could quickly compile a new motion planning application by simply choosing which algorithm to use for each functionality.**

## I. INTRODUCTION

In robotics software engineering we are observing an increasing need for composing new software applications out of reusable building blocks. Software reuse allows researchers to focus on their core problems instead of constantly re-writing each other's code. For example, experts in motion planning could experiment new path planning algorithms for a mobile robot relying on reusable implementation of obstacle avoidance and self-localization functionalities. Reuse of consolidated and shared implementation of common functionality allows different teams to test their new algorithms on common benchmarks in order to assess performance objectively.

Today, a huge corpus of software applications, which implement the entire spectrum of robot functionality, algorithms, and control paradigms, is available in robotic research laboratories and potentially could be reused in many different applications. For various reasons, the interoperability between different frameworks or their extentions towards novel applications is limited or would require high efforts.

The work described in this paper has been funded by the European Community's Seventh Framework Programme (FP7/2007-2013) under grant agreement no. FP7-ICT-231940-BRICS (Best Practice in Robotics).

D. Brugali and L. Gherardi are with the Dept. of Information Technology and Mathematics, University of Bergamo, 24044 Dalmine, Italy brugali@unibg.it, luca.gherardi@unibg.it

W. Nowak, A. Zakharov and E. Prassler are with GPS GmbH, Nobelstr. 12, 70569 Stuttgart, Germany {nowak, zakharov, prassler}@gps-stuttgart.de

In this paper we report on our experience in analysing and comparing open source implementations of best practice algorithms for motion planning in the context of the BRICS project (Best Practice in Robotics) funded by the European Commission. The domain of motion planning for mobile manipulators has made tremendous progress in the last years. Many problems are now well-understood and can be tackled with standard solutions [1]. Several sophisticated software frameworks exist. Unfortunately it is still rather difficult to compare them in an objective manner, due to significant variation in the software representation of common concepts (e.g. robot configuration, path, configuration space) and in the libraries' APIs. For this reason, in BRICS we have defined a methodology to harmonize motion planning libraries based on the concepts of software refactoring and software components.

Refactoring is the process of applying a series of small behavior-preserving transformations to an existing software system in order to improve its software quality. Component-based engineering is the state-of-the-art technology to develop modular, reusable, and interoperable software systems.

The set of guidelines we propose shows how to refactor existing software architectures in order to facilitate the integration of different implementations of algorithms in a common framework and to support the process of benchmarking different algorithms. Benchmarking is the process of assessing the relative performance of algorithms. But when algorithms are benchmarked, the algorithms are not compared themselves, instead two of their specific implementations. In order to obtain significant results it is of critical importance that the two implementations use the same data structures and differ only in details specific to the algorithms under consideration.

In this paper we report a specific example of application of these guidelines to the CoPP motion planning library, transfering it into the new component framework BRICS_MM. The transition is also put into context to other state-of-the-art motion planning libraries that have applied similar concepts to varying extents. However our guidelines are more general and can be used on every software architecture.

The paper is structured as follows.

Section II presents a survey of motion planning libraries, while Section III describes relevant aspects of their implementation related to interoperability issues. Section IV illustrates two software transformation patterns for refactoring motion planning libraries into software components. Section V presents the detailed design of the Path Planning component framework. Finally VI draws the relevant conclusions.

## II. Open Source Motion Planning Libraries

The focus in this paper is on path planning problems with a high number of degrees of freedom, as typically encountered in mobile manipulation tasks. In this context, several classes of probabilistic, sample-based planners have been evolved, most notably variants of Probabilistic Roadmaps (PRM) and Rapidly Exploring Random Trees (RRT), amongst others; see [1] for a comprehensive overview. These algorithms typically work in the configuration space (C-space) of the robot. Elements of the overall planning tasks (can) comprise [13]:

- representation of robot's configuration space
- representation of paths and trajectories
- kinematic or dynamic constraints
- sampling new points in the C-space
- measuring distances in the C-space
- interpolating between two points in C-space
- computation of forward or backward kinematics

- the global planner algorithm itself
- specification of start and goal conditions
- local planner for quickly connecting two configurations

- representation of the robot's geometry and the environment in the Cartesian 3D space
- collision checking
- updating path according to changing environments
- handling graph- or tree-like structures for roadmaps or discretisation of the C-space

In particular collision checking is known to play often a crucial role for sample-based planners, taking up most of the processing time. Many of the items mentioned above are addressed in current software libraries, in more or less explicit ways. An overview of these libraries is given below.

The Motion Strategy Library (MSL) [5] was developed by the research group of Steven LaValle at the University of Illinois. The library includes support for multiple planners (including variants of RRT, PRM and Forward Dynamic Programming FDP), collision checkers (PQP) and visualization in multiple formats. Originally deployed only under Linux, a Windows version was published in 2008.

The Motion Planning Kit (MPK) [6] was developed by the research group of Jean-Claude Latombe at Stanford University. It implements a fast single-query bi-directional probabilistic roadmap path planner with lazy collision-checking (SBL) and relies on PQP for collision detection.

The Motion Planning Kernel (MPK) [7] was first developed by Ian Gipson at the Computational Robotics Lab at Simon Fraser University. It comes with a full suite of collision detection algorithms (V-collide and SOLID amongst others) and implements path planners for RRT and PRM.

Components for Path Planning (CoPP) [9] was developed by Morten Strandberg at the Royal Institute of Technology (KTH) in Stockholm, Sweden, with the aim of a clearly structured object-oriented planning framework. Several functionalities such as samplers, metrics, local planners, interpolators, and collision checkers (PQP, YAOBI) are explicitly distinguished with separate base classes. The library includes planners for PRM, RRT and PCD and provides support for visualizing via Coin and VRML.

OpenRAVE [8] developed by Rosen Diankov is a framework that covers the whole development cycle around manipulation and grasping, including support for different sensor inputs, controllers and physical simulation. Plugins are meant to provide an easy way for users to add various custom functionalities. OpenRAVE integrates to ROS and has interfaces to Octave, Matlab and Python.

The Object-Oriented Programming System for Motion Planning (OOPSMP) [10] was developed at Lydia Kavraki's lab. It includes a large variety of motion planners and can handle kinodynamic constraints. Several general purpose data structures and functionalities for the domain of motion planning are provided.

Open Motion Planning Library (OMPL) [11] developed by Ioan Sucan stands out from the other libraries in the way that it explicitly concentrates on the core path planning algorithms. Other elements such as collision checking, simulation or motion control are handled by integration into the ROS framework. OMPL provides various planners including RRT, EST, SBL and KPIECE, and an inverse kinematics solver (GAIK) based on Genetic algorithms.

KineoWorks is the only framework mentioned here which is not available as open source. Originally developed as Move3D [12] at LAAS, it was put into a product by Kineo. It is meant to provide a component-based architecture that supports easy integration into applications. But as it is not free, we will not provide further information here.

All of the libraries discussed above but KineoWorks are published as open source or are free for non-commercial use. Most of them are under active development, while only a few seem to be discontinued.

Notably all libraries above are written in C++. Some include scripting support for other languages or interfaces to tools such as Octave or Matlab. All of them offer some kind of 3D visualization, some also support for simulation and physics engines; with the special note of OMPL that relies on the ROS environment for all those aspects.

## III. Interoperability issues

Most of the libraries cannot be easily interchanged and it is rather difficult to compare individual algorithms between libraries. One of the reasons is that they rely on some base classes, which sometimes are very detailed or include certain dependencies and that cannot easily be replaced or changed. Thus it is difficult to plug one algorithm, including all relevant aspects, into some other piece of software. In addition, many of the more internal aspects of planning algorithms, such as samplers or metrics, may not be made explicit. In order to exchange them, the algorithms' source code would have to be changed. The dependency of base classes on external frameworks may also restrict the transfer of a library onto real robots possibly with embedded PCs and limited resources. It should be noted that in particular the newer libraries include several mechanisms and design aspects that aim at minimizing the before-mentioned problems.

Nearly all libraries provide some kind of support for different implementations of functionalities, most commonly in the form of inheritance from a base class. This holds true for the main planner classes, but for example also collision checking engines are nearly always made explicit and interchangeable.

As an example, MSL is built around the classes `Model` (representing kinematic and dynamic systems), `Geom` (geometric objects for collision checking), `Problem` (general class to represent aspects of a path planning problem), `Solver` (base class for path planning algorithms), `Scene`, `Render` and `GUI` for visualization. New functionality may be added by inheriting from these main classes. In this specific library, metrics and interpolators for example may be changed by overriding virtual functions in the `Model` class. OpenRAVE and OOPSMP in addition provide the concepts of plugins, where different kinds of functionalities may be attached from outside. Those plugins have to inherit from base classes as well, but can then be loaded during runtime from dynamic libraries.

In OpenRAVE a central class `EnvironmentBase` glues all parts together. This is a container for all other elements, including physics and visualization. Most elements refer back to this container, e.g. loading from XML, connecting robots with collision checkers, or drawing is handled via calls to `EnvironmentBase`.

In OOPSMP the composition and configuration of planning problems is done via customized XML files. A parser translates the XML elements into calls to dynamic libraries. That way OOPSMP can be flexibly configured without touching any source code. On the other hand some kind of dedicated plugin functionality is needed to extend it. Similar to OpenRave, one container class `CoreRobotData` includes pointers to all components such as workspace, state space and smoother, with these components inheriting from `CoreRobotData`.

In contrast to the previous frameworks, OMPL is inherently integrated into ROS. The environment representation for the collision detector can be provided at runtime by an appropriate ROS node. The representation of the robot is loaded from URDF files. When paths have been planned, they are published to the ROS network. OMPL implements a number of abstract base classes such as `Planner`, `Path`, or `Goal`.

A major step concerning interoperability has been made in the ROS project. There a plenitude of standard interfaces for various aspects, from trajectories to robot kinematics and environment modeling, have been introduced in a data-centric way, without unnecessary functional dependencies. The libraries OpenRAVE and OMPL can be used over those interfaces, increasing the interoperability significantly. That way they come close to the ideas of refactoring as presented in section IV and V.

In the following we provide an overview on how some of the key concepts for motion planning tasks are implemented in the different software libraries. These includes data structures to represent *Configuration* (Table I), *Path* (Table II), *C-Space*, (Table III), *Robot Kinematics* (Table IV), and functionality such as *Metric*, *Interpolator*, *Sampler* (Table V), *Collision Detection*, and *Environment Modeling* (Table VI). Many of these concepts are represented in semantically similar ways. But the remaining differences constitute several of the major problems concerning interoperability between libraries.

TABLE I
CLASSES FOR POINT IN C-SPACE

| Library | Main class | Notes |
|---|---|---|
| MSL | MSLVector | `double` array, includes size |
| MPK$_{Kernel}$ | Configuration | `vector <double>`, includes calls to OpenGL |
| CoPP | Config | `vector <double>` |
| MPK$_{Kit}$ | mpkConfig | `vector <double>`, includes various functions |
| OpenRave | TPOINT | `vector <dReal>`, includes, velocities and time |
| OOPSMP | State_t | `double` array |
| OMPL | State | `double` array |

TABLE II
CLASSES FOR PATH OR TRAJECTORY

| Library | Main class | Notes |
|---|---|---|
| MSL | Planner | `list<MSLVector>`, located directly in planner's base class |
| MPK$_{Kernel}$ | PA_Points | `vector<Configuration>`, includes calls to OpenGL |
| CoPP | Path | `list<Cinfigurations>`, includes time |
| MPK$_{Kit}$ | sblPlanner | `list<mpkConfig>`, includes various functions |
| OpenRave | Trajectory | `vector<TPOINT>`, `vector<TSEGMENT>`, includes elements for dynamic motion control |
| OOPSMP | Path | includes interfaces for time, splitting and more. Base class with various implementations |
| OMPL | Path | points to a `SpaceInformation`, derived classes include array of `State` |

TABLE III
CLASSES FOR C-SPACE

| Library | Main class | Notes |
|---|---|---|
| MSL | Problem, Model | includes upper/lower limits, start and goal configuration. control inputs and system simulation |
| MPK$_{Kernel}$ | Universe, RobotBase | includes upper/lower limits, start and goal configuration |
| CoPP | DOF_Properties | stored in multiple places, where needed |
| MPK$_{Kit}$ | | limits are implicitly in planner |
| OpenRave | ConfigurationState | includes limits and number of DoF |
| OOPSMP | StateSpace | includes bounding box and various other functions, many concrete implementation |
| OMPL | SpaceInformation | includes start and goal configurations, dimension, `StateDistanceEvaluator`, `StateValidityChecker` |

TABLE IV

Robot Kinematics data structures

| Library | Main class | Notes |
|---|---|---|
| MSL | Model | includes kinematic structure and control |
| MPK$_{Kernel}$ | RobotBase | `vector <LinkBase*>` |
| CoPP | KinematicNode | `vector <DOF_Properties>`, limits are stored in Robot class as well |
| MPK$_{Kit}$ | mpkBaseRobot | includes pointer to a parent joint, spatial transforms, triangulated link model, PQP and SoQT data |
| OpenRave | KinBody | `vector vector<Joints>` `vector <Links>` |
| OOPSMP | StateSpace | implicitly defined via `StateSpace` and related classes |
| OMPL | | based on ROS using URDF files |

TABLE V

Interfaces for Metrics; Interpolator; Sampler

| Library | Notes |
|---|---|
| MSL | `Model` (and `Problem`) with virtual functions for Metric and Interpolator. Sampling as virtual function `ChooseState` in each planner class |
| MPK$_{Kernel}$ | Sampler and metric as virtual functions in planner base classes. Interpolation hard-coded in planner |
| CoPP | classes `Metric`; `Interpolator`; `ConfigSpaceSampler` |
| MPK$_{Kit}$ | non-virtual functions in class `mpkConfig` for metrics and interpolating. Sampling hard-coded in planner |
| OpenRAVE | classes `DistanceMetric`; `SampleFunction`; four interpolation methods hard-coded in `Trajectory` |
| OOPSMP | distance function in `StateSpace`; classes `PathGenerator`; `ValidStateSampler` |
| OMPL | classes `StateDistanceEvaluator`; `StateSamplingCore`; interpolation done in planners |

TABLE VI

Interfaces for Collision detector and environment modeling

| Library | Notes |
|---|---|
| MSL | `Geom` with derived class for PQP |
| MPK$_{Kernel}$ | `CollisionDetectorBase`. `Universe` has an array of `Mesh` which can model various objects. |
| CoPP | `ObjectSet`. Base class `Geom` stores a position, with inherited classes for triangles and convex objects. |
| MPK$_{Kit}$ | `mpkCollDistAlgo` uses PQP or own collision detector |
| OpenRAVE | `CollisionCheckerBase`. `KinBody` includes `TRIMESH` and `GEOMPROPERTIES` for modelling triangle meshes |
| OOPSMP | `CollisionDetector`. `Workspace` holds list of `Part`, support of polygons |
| OMPL | Based on ROS with interfaces of `CollisionSpace` and various geometry messages |

## IV. Refactoring towards Components

”A software component is a unit of composition with contractually specified interfaces and explicit context dependencies only. A software component can be deployed independently and is subject to composition by third parties.” [2]

Software components come with well-defined *component specifications*, which are abstractions from the details (data structures and operations) of their (possibly many) implementations. A component specification explicitly declares which functionality (*provided interfaces*) are offered to its clients, the public obligations (*contracts*) with its clients in the form of various kinds of constraints (e.g. preconditions, postconditions, invariants) on how to access the functionality, and the dependencies (*required interfaces*) to the functionality that are delegated to other components.

A *component implementation*, on the other hand, defines how the component supports those features and obligations in terms of a collaborative structure of realizing objects (class instances) and algorithms implementing the functionality declared in the component specification.

Separating the specification of components from their implementation is desirable for achieving modular, interoperable, and extensible software and allows independent evolution of client and provider components. If client code depends only on the interfaces to a component and not on the component's implementation, a different implementation can be substituted without affecting client code. If a coherent set of *required* interfaces can be defined that specify the most frequently used robot services and capabilities, and if robotics applications are designed around those interfaces, then every component implementing compatible *provided* interfaces has the potentiality to be reused in those applications.

The transition from a motion planning class library such as CoPP to a motion planning component system has required the redistribution of responsibilities (i.e. functionalities) among the original classes and added new classes. In addition, it involves the aggregation of class instances (objects) into components and the definition of component interfaces that make components functionality available to their clients. In particular, the refactoring process of the motion planning library consisted of the ordered application of the following four well-known architecture refactoring patterns [3], which provide concrete guidelines to restructure the architecture of software systems: *Move Behavior Close to Data*, *Split up God Class*, *Eliminate Navigation Code*, and *Transform Conditionals into Registration*. We describe the first two transformations in the following as they are mostly related to the definition of components interfaces, and the remaining two in next section as they are mostly related to components implementation.

### A. Transformation I: Move Behavior Close to Data

We have first analyzed the data structures used in the various motion planning libraries, cf. section III, in order to identify similarities. As expected, nearly all the algorithms rely on the concept of robot configuration, but the data structures used to represent them show subtle differences. While an array or vector of double values is used in all cases, they differ concerning additional information such as time and flags, and whether the number of dimensions is included or not. For some versions it is trivial to convert between each other, while for others it is a problem due to different kind of information stored.

The refactoring pattern *Move Behavior Close to Data* suggests introducing data containers that harmonize existing data structures and assigning them the responsibilities to create, initialize, update, transform, and elaborate encapsulated data.

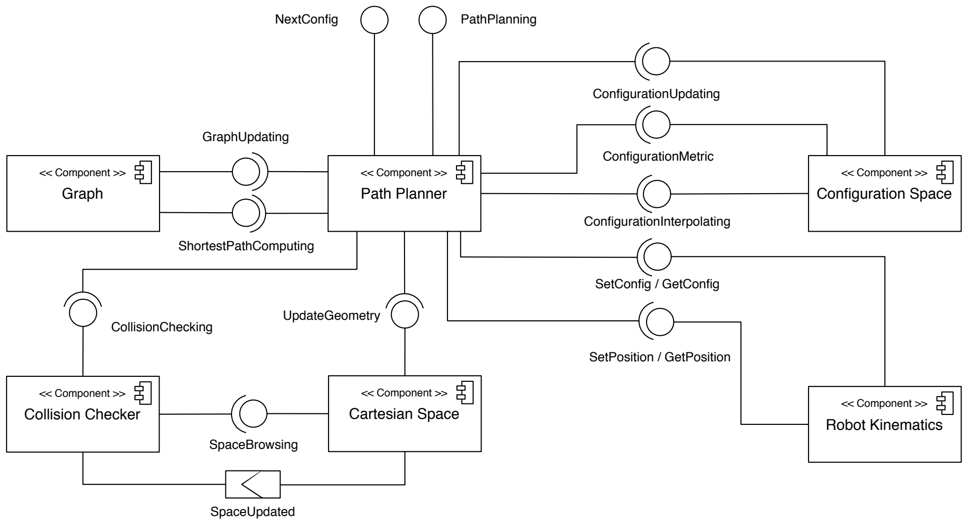

Fig. 1. The components system resulting from refactoring motion planning libraries

The guidelines of this pattern have lead us to the definition of the new *ConfigurationSpace* component (see Fig. 1). It is in charge of maintaining the representation of the robot configuration space. *Configuration* is represented as a vector of n values, each representing a point in the robot's configuration space. The *ConfigurationSpace* component stores at least the following configurations:

- *lowerConfig* is the configuration that represents the lower bound of the configuration space
- *upperConfig* is the configuration that represents the upper bound of the configuration space
- *currentConfig* is a list of robot's configurations at given instants of time.

Depending on the type of configuration space, additional data may be required for example to represent rotational joints or a rigid body moving in 2D space, resulting in $SE(2)$.

The *ConfigurationSpace* component offers services that were implemented as separated class hierarchies in the original motion planning libraries, such as the algorithms for configuration interpolating and for measuring distances between pairs of configurations. The configuration pair may be provided by the client or may correspond to two current configurations at different instants of time. We have defined the following three provided interfaces:

- *ConfigurationSetup*
- *ConfigurationInterpolating*
- *ConfigurationMetric*

The *ConfigurationSpace* component does not implement any required interface. Thus it has no dependency to other components and can be reused independently of other motion planning components as building block for the implementation of robot functionality where robot's configurations need to be represented, such as motion control, navigation, and manipulation.

We have applied the refactoring pattern *Move Behavior Close to Data* to the original motion planning libraries iteratively. It has lead us to the identification of two more components that behave as data container with provided interfaces only (see Fig. 1):

- The *CartesianSpace* component encapsulates the geometric representation of the robot's environment and implements interfaces for updating and browsing it.
- The *Graph* component is a wrapper of external graph management libraries that implements standard interfaces for creating, updating, and processing data organized as graph structures.

## *B. Transformation II: Split up God Class*

Motion planning algorithms are often implemented by structuring the code according to the functional decomposition approach, where most of the logic of the functionality is provided by a single "god class", like *MotionPlanner*. God classes are hard to extend, modify or subclass because they assume too many responsibilities and changes affect large numbers of methods or instance variables. The *Split up God Class* pattern refactors a procedural god class into a number of simple, more cohesive classes.

The iterative application of this pattern to the motion planning class libraries has generated three components: *RobotKinematics*, *CollisionChecker*, and *PathPlanner*. The first iteration has produced the clear separation of two core functionalities, i.e. collision checking and path planning. Most class libraries already offer distinct specialization hierarchies for the implementation of collision checking and path planning algorithms, but their high level abstract classes are incompatible and in some cases have a long list of methods with a large number of parameters.

The *CollisionChecker* component maintains an internal representation of the robot environment, which is an algorithm-specific approximation (e.g. using bounding boxes) of the Cartesian space. This internal representation needs to be updated when the robot's Cartesian space gets modified, for example when the robot or other objects change their position. For this purpose, this component requires the *SpaceBrowsing* interface of the *CartesianSpace* component and implements the *SpaceUpdated* event listener. The provided interface *CollisionChecking* defines the operations that the path planner can invoke to check and inspect collisions among objects in the robot's environment.

The *RobotKinematics* component stores the robot's kinematic model and implements only provided interfaces for invoking the forward and inverse kinematic transformations.

Finally, the *PathPlanner* component implements the algorithms that generate a robot path as a sequence of collision-free configurations. The simplified interaction between the seven components in Fig. 1 consists of the following sequence of steps: *PathPlanner* generates (samples) a new robot configuration, updates *ConfigurationSpace*, gets the new robot position from the *RobotKinematics* component, updates *CartesianSpace*, checks if the new configuration is collision free and, if this is the case, updates *Graph*.

Thus, it is clear that the *PathPlanner* component uses and integrates the services of the other components to build a specific robot functionality and that these components can be reused as building blocks for the implementation of other functionality.

## V. Path Planning Component Framework

The separation of interface and implementation facilitates component interchangeability: a component can be replaced with another one that implements the same provided interface. The various implementations of a component may differ in functional characteristics (i.e. different algorithms for motion planning), non-functional properties (i.e. performance, maintainability, documentation quality, reliability), realizing technology (e.g. the description of the geometric space may be stored in a relational database or as XML files) and even programming language (if components are build on a middleware or multi-language run-time infrastructure).

Despite of these differences, components that implement the same interfaces and offer similar functionality are typically implemented around common entities and mechanisms, which are core aspects of the provided functionality (e.g. the concepts of *Path* and *Configuration* in motion planning) and can be represented as stable data structures and operations. In contrast, those aspects of a component implementation that are more likely to be affected by the evolution of the application domain represent its variation points.

Component frameworks enable a clear separation between stable and variable aspects of a component implementation. A component framework is a skeleton that can be specialized to produce custom components. As such it represents a family of component implementations, which can be derived from its design and built on its data structures and operations without changing them.

In this section we describe how we have refactored the components depicted in Fig. 1 into corresponding component frameworks. In particular, we illustrate the *PathPlanner* component framework depicted in Fig. 2, which clearly separates stable data structures (black boxes), variation points (blue boxes), and concrete variants (red boxes).

The classes used to represent a robot path are stable entities of the component framework. They have been structured according to the *Composite* design pattern [4]. *PathLeg* is a sequence of *Configuration* objects. Let us consider a mobile manipulator that navigates inside a building. A path leg may correspond to the sequence of configurations of the mobile platform from a place inside a room to a place close to the door. Another path leg may then correspond to the sequence of configurations of the manipulator to open the door. *CompositeLeg* is a composition of path legs. The *Composite* design pattern allows the hierarchical composition of even more complex paths, which can be browsed through a uniform interface implemented by the *Path* abstract class.

The variation points are abstract classes that implement stable data structures and operations that are common to a family of similar algorithms. From the analysis of motion planning libraries we have identified four core variation points: *GlobalPlanner*, *LocalPlanner*, *Sampler*, and *PathUpdater*. The component developer customizes the component framework by supplying concrete subclasses (e.g. *PRMPlanner*, *BinaryConnector*, *UniformSampler*, and *ElasticStripUpdater*) that implement specific algorithms and represent possible variants of each variation point.

After stable data structures, variation points and variants had been identified, we refactored the motion planning library according to the two refactoring patterns described in the following sections.

### *A. Transformation III: Transform Conditionals into Registration*

Components frameworks can be customized at design time, when the software developer implements specific variants (e.g. algorithms) for each variation point, or at run time, when one of several alternative variants is selected according to current execution context. For example, variants of a specific family of algorithms could be switched through a graphical user interface in order to benchmark and compare their performance during experiment sessions. Alternatively, the robot could select the most effective algorithm autonomously according to situation awareness (e.g. a fast path planner in open space environments and a powerful path planner in cluttered environments).

Both situations require a component's client (e.g. the GUI or the robot controller) to switch among several variants. This is potentially implemented as long methods consisting almost entirely of case statements, which make the code more difficult to maintain.

The pattern *Transform Conditionals into Registration* aims at reducing the coupling between component variants and clients so that the addition or removal of variants does not lead to changing the code of the clients. Therefore the pattern suggests introducing a registration mechanism to which each variant is responsible for registering itself. The component clients are then transformed to query the registration repository instead of performing conditionals.

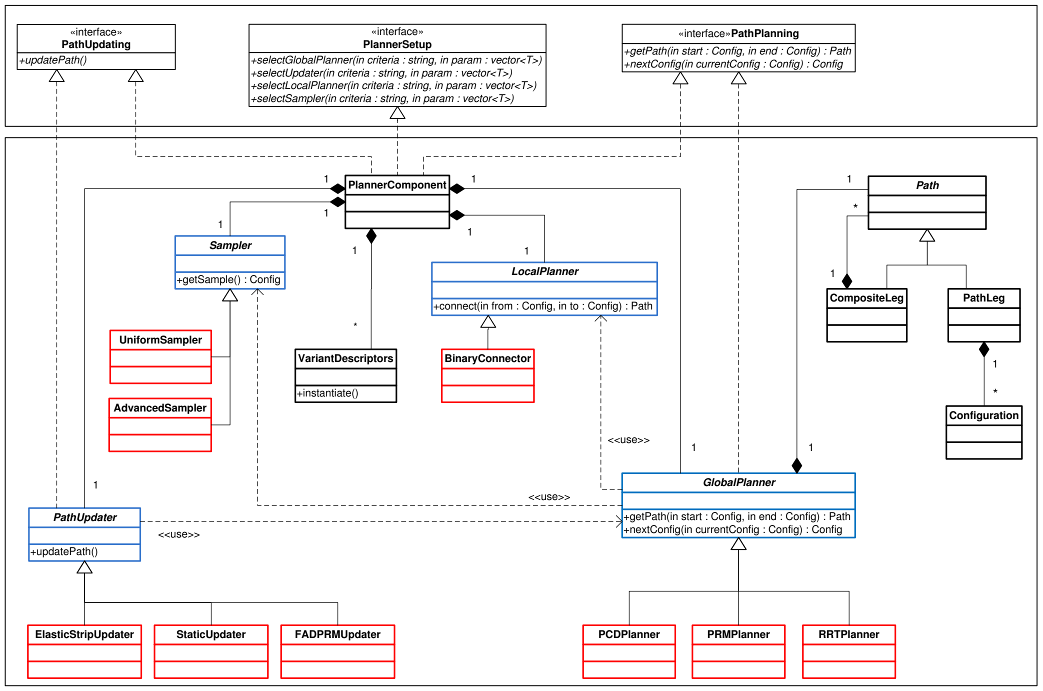
«interface»
PathUpdating
+updatePath()
«interface»
PlannerSetup
+selectGlobalPlanner(in criteria : string, in param : vector<T>)
+selectUpdater(in criteria : string, in param : vector<T>)
+selectLocalPlanner(in criteria : string, in param : vector<T>)
+selectSampler(in criteria : string, in param : vector<T>)
«interface»PathPlanning
+getPath(in start : Config, in end : Config) : Path
+nextConfig(in currentConfig : Config) : Config
PlannerComponent
Sampler
+getSample() : Config
LocalPlanner
+connect(in from : Config, in to : Config) : Path
UniformSampler
AdvancedSampler
VariantDescriptors
+instantiate()
BinaryConnector
Path
CompositeLeg
PathLeg
Configuration
<<use>>
GlobalPlanner
+getPath(in start : Config, in end : Config) : Path
+nextConfig(in currentConfig : Config) : Config
PathUpdater
+updatePath()
ElasticStripUpdater
StaticUpdater
FADPRMUpdater
PCDPlanner
PRMPlanner
RRTPlanner

Fig. 2. The Path Planner Component Framework

We defined class *VariantDescriptor* that encapsulates the information necessary for registering, querying, instantiating, and using each component variant and class *VariationManager* that is queried by the component clients to check the presence of and instantiate specific variants. For each variation point (e.g. *GlobalPlanner*) class *PlannerComponent* encapsulates a member variable that points to the current selected variant.

## B. Transformation IV: Eliminate Navigation Code

A component's clients need to access functionality, which in most cases are provided by specific variant objects. For example, the *GlobalPlanner* variation point represents the core logic of the *PathPlanner* component and is available in several variants, each one implementing a specific algorithm for global planning. The *GlobalPlanner* and its variants implement interface *PathPlanning*, which defines the fundamental operation `Path getPath(Configuration start, Configuration end)`.

The analysis of the motion planning libraries described in Sections II and III reveals that clients of these libraries (e.g. the robot control application) typically have direct access to the objects that implement planning algorithms. In our case, direct access to variant objects would require navigating through classes *PlannerComponent* and *VariationManager* in order to get a reference to individual variant objects. This would violate component encapsulation and would couple clients and variant objects unnecessarily.

Pattern *Eliminate Navigation Code* suggests preventing these problems by transforming object containers into service providers. This is the case of *PlannerComponent*, which maintains pointers to current variant objects. It implements interface *PathPlanning* and delegates the execution of its operations to the current variant of *GlobalPlanner*.

By applying Transformation III and Transformation IV, we defined two distinct provided interfaces for the *PathPlanner* component. Interface *PlannerSetup* allows clients configuring the component by selecting specific variants for each variation point. Interface *PathPlanning* allows clients accessing component's functionality. It is clear that different clients can access the two interfaces independently. For example, the GUI (one client) can switch two variants of the same variation point (e.g. *LocalPlanner*) that will be used to compute paths for the robot controller (another client).

## VI. CONCLUSIONS

Economic efficiency and competitiveness as well as scientific and technical quality create an increasing pressure on robot software engineers to refrain from so-called "from scratch" and "me too" developments of robotic software but instead refer to existing, reusable software components. Unfortunately the notion of reusability has gained only limited attention in the robotics software developer community so far.

In the BRICS project we have been developing a methodology that shall make off-the-shelf software libraries for robot functionalities reusable. In this paper we described an approach to refactor existing object oriented software into reusable software components developed in this project. We presented guidelines in section IV and V in form of four transformation steps that aim at increasing the reusability of software in the domain of motion planning.

Although very sophisticated libraries have been created for this task in the last years, a focus on object-oriented designs puts limits to reusability and interoperability. In addition, ideas similar to the presented refactoring steps have often not been made as explicit. We have actually applied these steps to the CoPP library, which was designed to foster a good object-oriented design. These steps, together with various other extentions, have lead to the component framework BRICS_MM which is available as open source from the website `www.best-of-robotics.org`.

The work presented in this paper is ongoing work. Although the benefits of software reuse may appear obvious, we still lack objective, measurable and agreed indicators which demonstrate the use and the economic and scientific gain of reuse.

The BRICS project will undertake a significant effort to identify such objective and measurable indicators which are agreeable by the community of robotics software engineers and developers.

## VII. ACKNOWLEDGMENTS

The authors would like to thank all the partners of the BRICS project for their valuable comments.

## REFERENCES

[1] Steven M. LaValle. *Planning Algorithms*. Cambridge University Press, 2006.

[2] C. Szyperski. *Component Software: Beyond Object-Oriented Programming*. Reading, MA: Addison-Wesley, 2002.

[3] S. Demeyer, S. Ducasse, O. Nierstrasz. *Object-Oriented Reengineering Patterns*. Morgan Kaufmann, 2008

[4] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Reading, MA: Addison-Wesley, 1995.

[5] S. LaValle, P. Cheng, J. Kuffner, S. Lindemann, A. Manohar, B. Tovar, L. Yang, and A. Yershova. "MSL - Motion strategy library". http://msl.cs.uiuc.edu/msl/

[6] J.-C. Latombe, F. Schwarzer, and M. Saha. "MPK - Motion Planning Kit". http://robotics.stanford.edu/˜mitul/mpk/.

[7] I. Gipson, K. Gupta, and M. Greenspan. "MPK: An open extensible motion planning kernel". *Journal of Robotic Systems*, Volume 18, Issue 8, pp 433 - 443, 2001. http://ramp.ensc.sfu.ca/mpk/

[8] Rosen Diankov and James Kuffner. "OpenRAVE: A Planning Architecture for Autonomous Robotics". *Tech. Rep. CMU-RI-TR-08-34.* Robotics Institute, Carnegie Mellon University, Pittsburgh, PA, July 2008. http://openrave.programmingvision.com

[9] M. Strandberg. "Robot path planning: an object-oriented approach". *Doctoral thesis*, Royal Institute of Technology (KTH) Stockholm, Sweden, October 2004. http://sourceforge.net/projects/copp/

[10] E. Plaku, K. E. Bekris, and L. E. Kavraki. "OOPS for motion planning: An online, open-source, programming system." *In Proc. of the IEEE International Conference on Robotics and Automation (ICRA)*, April 2007. http://www.kavrakilab.org/OOPSMP/index.html

[11] Ioan Sucan. "OMPL - Open Motion Planning Library". http://www.ros.org/wiki/ompl

[12] T. Simeon, J. P. Laumond, and F. Lamiraux. "Move3D: A generic platform for path planning". *In Proc. of the IEEE International Symposium on Assembly and Task Planning*, May 2001.

[13] Ioan A. Şucan and Lydia E. Kavraki. "On the Implementation of Single-Query Sampling-Based Motion Planners". *IEEE International Conference on Robotics and Automation (ICRA), May 2010.*